Tutorial: Open Source Enterprise Application Integration

Introducing the Event Processing Capabilities of Apache Camel

Christoph Emmersberger
Universität Regensburg
Universitätsstraße 31
93053 Regensburg
christoph@emmersberger.org

Florian Springer
Senacor Technologies AG
Wieseneckstr. 26
90571 Schwaig b. Nürnberg
florian.springer@senacor.com

ABSTRACT

"Interesting applications rarely live in isolation." ([1], xxix) With this sentence G. Hohpe and B. Woolf start the introduction to their book *Enterprise Integration Pattern: Designing, Building, and Deploying Messaging Solutions*.
While the statement is valid now for more than ten years, Gartner estimates today the cost increase targeting integration aspects for midsize to large companies at about 33% within the next three years (cf. [2]). The expected increase will be mainly driven by the integration of cloud services and mobile devices. Since event processing addresses clearly problems arising with the growth of computational distribution, particularly with the increasing number of mobile devices or cloud services, integration is a topic that needs to be addressed by event processing functionalities.
One of the frameworks within the integration domain is *Apache Camel*. Since it's initial release in 2007, the framework has gained quite some attention - not only within the open-source arena. *Apache Camel* has a strong focus on enterprise application integration since it implements well known *Enterprise Integration Patterns (EIP's)* (cf. [1]).
This work reveals the event processing capabilities of *Apache Camel* alongside a logistics parcel delivery process. The delivery process facilitates the scenario descriptions to exemplify the event processing functionalities within a real-world context. All coding examples, supporting the functionality demonstration, are setup around the shipment of parcels.

General Terms

Software Engineering

Keywords

Integration Framework, Event Processing

1. INTRODUCTION

"Enterprise Application Integration is the creation of business solutions by combining applications using common middleware." ([3], 2) Middleware specifies a technology stack which is capable to mediate between applications with the overall goal of improving the supply-chain relationships in a distributed application environment. One of the frameworks supporting the mediation between applications coming from the open-source domain is the integration framework *Apache Camel*. The first version of *Camel* has been released in version 1.0 the 2nd July 2007 - just three and a half month after the development had started (cf. [4]).
Regardless of the short development period, the first release covered already two domain specific languages (DSL's) based on Java and XML, the core routing functionality, an initial set of components, examples and a proper project setup [5]). Since than, *Camel* has become an Apache top level project in January 2009. Beside those organizational changes, the project has faced continuous growth which can be exemplified by

- the growing number of committers (starting at seven being a group of more than 30 today),
- an increasing code base which is today more than ten times the size of the first release and
- the growing number of components from an initial set of 18 to todays 140 components (including external components) listed at the *Camel* website (cf. [6]).

In August 2009, *Camel* has experienced a major release (cf. [7]). Since that time, it is available in version two which is still actively maintained. Having said that *Camel* is a middleware technology and "(...) middleware can be regarded as containing the roots of a hierarchical approach to events and event processing (...)" ([8], 37), we can also identify a clear relationship between enterprise application integration and event processing.

Within this tutorial paper we introduce *Apache Camel's* event processing capabilities. The first section gives a brief introduction into the context of a parcel delivery process. The process description does not claim to be generally valid; it's aim is furthermore to serve as consistent basis for all use case descriptions when characterizing the individual event processing functions and their corresponding *Camel* implementation.
After having a rough overview about the supply-chain interactions, the second section brings event processing and it's functionalities into the focus of this work. In this connection we'll cover the the topic's: *"Event type and event object"*, *"event producer, consumer and channel"*, the *"event processing network"* and *"event processing agents, context and state"*.

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

The subsections *event processing network*, *event producer, consumer and channel* and *event type* provide an introduction into the concepts of *Apache Camel* while creating cross references to event processing *building blocks* ([9], 59). The section on *event processing agents, context and state* will be covered in a software pattern like structure (cf. [1], xli and [10], 7). Since it is unquestionable a topic of it's own, setting a well defined structure for a pattern, we do not intend to impose any standardization along that line. Instead the pattern like descriptions collates the commonalities of both structures to describe event processing agent functionalities in a consistent manner.

Name identifies the event processing function. The intention of the name is also to provide a brief summary of the described functionality.

Context provides background information on requirements the event processing function aims to resolve. The context information is grouped around the parcel delivery process (cf. chapter 2)

Problem summarizes the problem that is being addressed by the event processing function in general without any contextual description.

Solution describes how *Camel* addresses the problem conceptually. The solution is based upon a graphical representation and a textual description that explaining the solution design.

Example demonstrates the implementation of the solution design with the *Apache Camel* framework. The description contains listings and descriptive elements that explain the implementation approach.

Finally this paper concludes in chapter 4 with a summary of *Camel's* event processing capabilities and discusses possible extensions supporting today's event processing needs.

2. BACKGROUND: PARCEL DELIVERY

A common parcel logistics delivery process (cf. [11], 171), starting with a customer order and ending with reconciliation and invoicing activities, serves as industrial context on which we will discover different event processing functionalities and their corresponding implementation within *Camel*. The generalized process consists of *Order Management*, *Delivery Disposition*, *Delivery Execution* and *Order Reconciliation* (see figure 1).

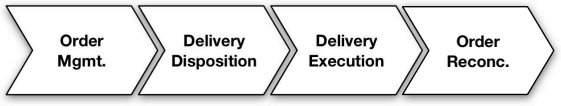

Figure 1: Parcel Delivery Process

Order Management enables the placement of transportation orders across multiple distribution channels. The distribution channel is e.g. an internet portal, an application installed at the customer location, a call center or even a mobile application. An electronic order triggers the physical pickup of an item before the delivery execution can start.

Delivery Disposition is the scheduling component within the parcel delivery process. Amongst others, it defines the size of the delivery area and schedules transportation routes. For distributing the scheduling information, the disposition step needs to integrate with the delivery execution, i.e. sorting sites, navigation systems and scan facilities.

Delivery Execution realizes the actual transport process encompassing *Pickup*, *Sortation*, *Transportation* and *Delivery*. At all stages, multiple devices with different hard and software configurations provide status information about an individual order, e.g. mobile scan devices during *Pickup* and *Delivery*, fixed installed scan systems during the *Sortation* process and positioning information while the *Transportation* is being conducted.

Order Reconciliation covers the invoicing and customer care activities. In case of any failures within the delivery process, customers may receive a refund and an insurance claim needs to be opened. Otherwise a monthly bill is created and the incoming payments will be monitored against payment transactions.

3. EVENT PROCESSING

Middleware is, beside other technologies, one of the established technology stacks which has clearly adopted the event processing paradigm ([8], 27-29 and 36-38). This section discovers *Camel's* implementation of the main event processing concepts ([9], 40-47) starting with *event types and objects*, highlighting the *event producer, consumer and channel* concept, integrating the components into an *event processing network* and finally discussing the interaction of *event processing agents, context and state*.

Event types and event objects Events are a representation for something which has happened. As a computational element for the technical representation, an event object is a concrete instance of an event type. All event objects, which belong to an event type, carry the same semantics within their object structure. Event types, objects and their implementation within *Camel* are covered in section 3.1.

Event producer, consumer and channel Producer and consumer are both event processing elements that communicate with each other. While an event producer obtains the role for generating events, the event consumer receives the results. An event channel fulfills the mediation role between producer and consumer. It's main task is to route events from the producer to the appropriate consumer. All three elements and their technical interaction are described in section 3.2.

Event processing networks An event processing network combines all processing elements, notably *event producer* and *consumer*, *event channel*, *event processing agent* including *context* and *state* elements into an executable application unit. This does not necessarily imply that all elements need to be executed within a single execution node. In section 3.3 you'll find an in depth explanation on how the combination can be achieved within *Camel*.

Event processing agents, context and state An agent is a software component with the purpose of processing events. The main functionalities of an agent can be summarized as *filtering*, *matching* and *derivation* with support of *context* and *state* information. While *filtering*, *matching* and *derivation* are core functions within event processing, the processing of *context* and *state* is required for event reasoning which means the detection of any conditions that lead to the event generation (cf. section 3.4)

3.1 Event Types and Event Objects

An event is a representation for something which has happened in reality (cf. [12], 151 and [8], 255-256 and [9], 4). Considering the parcel delivery process of figure 1, the central event which triggers all subsequent activities is a customer order for the transportation service, also known as *Shipment*. While the actual instance of an event is called *event object*, the structural definition is captured within an *event type*.
Listing 1 and 2 disclose the relationship between *event type* and *event object*. The *event type* in listing 1 defines, that a shipment order consists of *ShipmentDetails*, *Sender* and *Receiver*. *ShipmentDetails* describes the nature of the *Shipment*, particularly the size and weight of the item; *Sender* and *Receiver* specify the geographical nodes and the distance that needs to be covered by the *Shipment*.

Listing 1: Event Type - Shipment Order

```
<?xml version="1.0"?>
<xsd:schema version="1.0" ... >
  <xsd:element name="Shipment"
    type="tns:ShipmentType"/>
  <xsd:complexType name="ShipmentType">
    <xsd:sequence>
      <xsd:element name="ShipmentDetails"
        type="tns:ShipmentDetailsType"/>
      <xsd:element name="Sender"
        type="tns:SenderType"/>
      <xsd:element name="Receiver"
        type="tns:ReceiverType"/>
    </xsd:sequence>
  </xsd:complexType>
  ...
</xsd:schema>
```

In contrast to the *event type*, which describes the semantical structure of an event, the *event object* captures what is actual happening or has happened - in our example the concrete request of an order. The *event object* is therefore an instance of an *event type* which contains actual values within the structure of the event type. Listing 2 shows an instance of the *Shipment* with concrete values. While *Weight*, *Length*, *Width* and *Height* define the physical consistency of the Shipment, the *ParcelType* provides a classification information, in our case "Small Parcel". This classification information is not necessarily required, since it could be derived by some *context* information, e.g. when conducting a lookup against a context store. For the purpose of simplicity we decided to retain the attribute and discuss the topic of *context* in section 3.4.

Adding one more comment to the given XML listing: It is of course not an imperative need to specify an *event body* as XML structure. *Camel* supports any Java object type in its message body since the body is of type *"java.lang.object"* (cf. [13], 14). It is therefore possible to process any event that can be serialized into a Java object.

Listing 2: Event Object - Shipment Order

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:Shipment ... >
  <tns:ShipmentDetails>
    <tns:ParcelType>
      Small Parcel
    </tns:ParcelType>
    <tns:Weight>1.87</tns:Weight>
    <tns:Length>55</tns:Length>
    <tns:Width>25</tns:Width>
    <tns:Height>12.5</tns:Height>
  </tns:ShipmentDetails>
  <tns:Sender ... />
  <tns:Receiver ... />
</tns:Shipment>
```

Having introduced the abstract relationship between *event type* and *event object*, we need to investigate the logical structure of an event and it's attributes. Since the XML-based example above represents only the payload, which is usually wrapped in the *event body* within the context of event processing, we need to discover what other attributes are required to create an event. In addition we need to find out, if these elements find an implementation representation within the *Camel* framework.

According to ([9], 62-64), the abstract, logical structure of an event is split into three sections: An *event header*, a *payload* and an *open content* section. The following enumeration describes each element and it's role within event processing.

Header: Covers system defined event attributes e.g. a *type identifier*, a property that flags if it is an *event composition*, the *temporal granularity* and additional *event indicators* such as the *occurrence* and *detection time*, *event source*, *identity* and *certainty*. Header attributes support an efficient processing of the events since they reduce the need to lookup frequently used information.

Payload: Contains data attributes that are specified by the event type. The payload data is the actual, computational representation of what has happened (cf. listing 1 and 2).

Open content: Defines additional data that may be included in the even instance. This encompasses any binary attachment such as any document, audio or video file. However, it is also possible to attach structure or un structured text.

The logical event structure has its representation within *Camel* as *message* object, consisting of *headers*, a *body* and an *attachment* (cf. [13], 13). When taking a look at the *headers* object, we can identify a unique *messageId*. In spite of the *messageId* there exist no other predefined message headers and it is necessary to extend the research also to the surroundings of the *message object* to find additional information like the ones defined by the *event indicators*, e.g the *event source*.
In figure 2 we find an abstract illustration that explains how the message object is embedded within the broader context of an *exchange object*. An *exchange* may contain two message objects, an in and an out object. The reason for that

implementation can be found within the concept of *message exchange pattern (MEP)*. *Camel* knows basically two styles of exchanging messages: *InOnly* and *InOut* ([13], 14-15) . While some interaction scenarios manage the processing of a "fire and forget" style (i.e. *InOnly* pattern), others require still the information about the original request message (i.e. *InOut* pattern). To implement this behavior, the exchange contains the information about the pattern style and the option to store two messages.
In addition of the processing behavior, the exchange has also an attribute indicating the *endpoint* where the message camel from respectively the *event source* (cf. section 3.2). Beyond that, the concept involves also the knowledge about it's creation time (i.e. *properties.CamelCreatedTimestamp*), the execution context (i.e. *fromRouteId*) as well as the information if any processing failure. The advantage of carrying exception information outside of the actual message is, that the the exception handling needs only to look at this information rather than parsing the entire message body.
If it is required to add any additional event indicators, this can be handled by setting custom properties within the *message header* or the *exchange properties*.

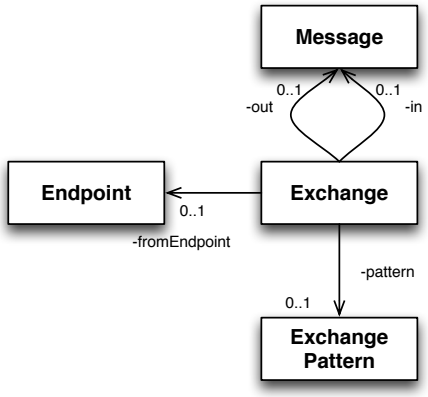

Figure 2: Camel Exchange and Message Model

Reviewing the overall *Camel* implementation it is fair say, that the message and exchange concept provides everything which required by an *event type and object* perspective to fulfill the needs required for event processing.

The next section introduces now the concepts of *event producer*, *consumer*, *event channel* and tries to discover their implementation within the *Camel* framework.

3.2 Event Producer, Consumer and Channel

Event producer and *consumer* are entities that interact with each other in an *event processing network* (cf. section 3.3) by sending and receiving events (cf. [9], 42). Taking an example from the parcel delivery process (cf. figure 1), a customer may submit a shipment order in a customer portal. The order is being sent to the order system which keeps track about the fulfillment degree of all customer orders (cf. figure 3). Before the order system is in capable to confirm that the order can be processed, it requires additional information about the feasibility from the scheduling system since the order system does not know anything about capacities and their utilization. The scheduling system responds upon the scheduling request and enables the order system to send out an order confirmation to the customer's inbox in the portal application.

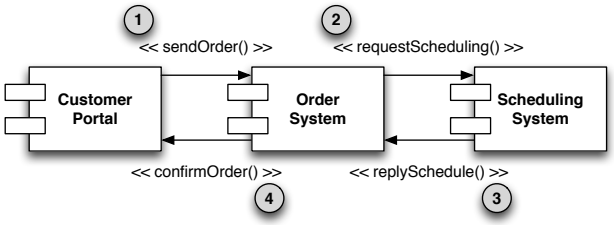

Figure 3: Order Submission

Under the condition of each *participant* acting either as *producer* or *consumer*, the example (cf. figure 3) demonstrates that each participant can obtain multiple roles depending upon the communication direction. At first, when sending the order request, the customer portal holds the event producer role before it turns into an event consumer when finally receiving the order confirmation. Since the role change does not affect any internal semantics or structure of data within the customer portal, it is possible to combine both roles into a single element, called *participant* (cf. [9], 33 and [14], 298). The *Camel* implementation of a participant element is realized via the so called *Component* object (cf. [13], 188-236).

"Components are the primary extension point in Camel" ([13], 189) and implement basically an endpoint factory. Since any *endpoint* is capable to send and receive events, a *component* exposing an endpoint is suitable to realize an *event producer* as well as a *consumer*. What might become a surprise is, that the generic *Component* concept is also capable to realize the concept of an *event channel*. First of all, an event channel is capable to receive events, similar to the behavior of an *event consumer*. Secondly, it acts like an *event producer* when sending events to one or more destinations. Finally, an *event channel* may also modify an input event or make routing decisions (cf. [9], 189).
Since a *Camel Component* provides basically a configurable endpoint to which someone can send events or may retrieve events, it is possible to cover the first and second statement. In addition the endpoint configuration may also contain information on how to modify (e.g. change header information) or apply routing decisions. Figure 4 provides an overview about the object dependencies in *Camel* between, *producer*, *consumer*, *component* and *endpoint*.

Component is a factory for *endpoint* objects. Components can be added to an *EPN* via configuration and inclusion to the *Camel Context* (cf. section 3.3).

Endpoint realizes an addressable element that can send and receive event objects. The endpoint address is specified as *Unified Resource Identifier (URI)* (cf. [15]).

Producer provides a channel on which clients can send event objects in an *endpoint*. The endpoint needs to be individually configured.

Consumer consumes events from an *endpoint*. Consuming events requires an individual configuration, including the appropriate endpoint addressing.

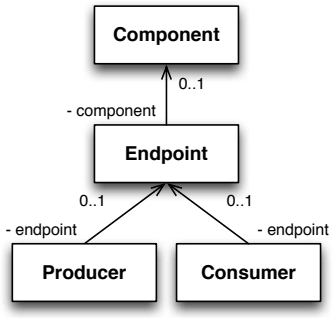

Figure 4: Abstract Component Model

Having gained a first insight into the *component* concept and its adoption to realize *event producer*, *consumer* and *channel* we'd like to give you an example that should provide some clarification. As example, we have selected two implementations within *Camel* that act like *event channels*, since a channel realizes both communication directions.
The first component is called *SEDA-component* and realizes the concept of a *staged event-driven architecture* endpoint (cf. [16]). This endpoint implements in our example (cf. listing 3) an endpoint that can receive events. Nevertheless, a SEDA endpoint may be also used to send events; it can therefore act in both *consumer* and *producer* role. In addition, the component has also the capability to configure basic routing rules such as the enablement of multiple consumers, similar to a topic based communication, as well as some blocking and sizing parameters.
The second component we have decided to use as an event channel is the *JMS-component*, since the Java Message System (JMS) ([17]). is a well known implementation of the *event channel* concept.

Listing 3 shows the mediation between two systems. The first system, in our case the order system, is capable to call a SEDA queue directly while the second system, responsible for scheduling and routing, can only understand JMS messages (cf. figure 3). The communication direction, respectively the indication of which component acts as *event producer* or *consumer*, is described by the *Camel DSL*, a processing language with simple routing expressions. In our example it is a simple *"from().to()"* clause based upon the fluent builder concept (cf. [13], 30, 132).

Listing 3: Endpoint URI Examples

```
from("seda://shipmentOrder" +
      "?multipleConsumers=false")
  .to("jms:topic:schedule?transacted=true");
```

As already stated, a component can be addressed via an *URI* scheme. In contrast to the *URI* specification (cf. [15], 16-25), *Camel* implements a simplified *URI* structure (cf. [13], 19, 25) to configure and address an *endpoint* individually (cf. listing 3). A *Camel* endpoint URI is based upon a *scheme*, a *context path* and *options*.

Scheme: The scheme references the component that needs to be instantiated. The component identifier needs to be uniquely.

Context path: Identifies the resources within the processing context unique. This is required to address the individual component instance when calling an endpoint.

Options: Options configure the endpoint behavior. The set of options is different for each component since the behavior depends upon the underlying component provider (e.g. the scheduling system)

Having seen now, how *Camel* realizes *event types* and *event objects* (3.1), and knowing how *event producer*, *consumer* and *channel* (3.2) are implemented and interact within the framework, it is time to discover the complete interaction of all elements within the *event processing network*.

3.3 Event Processing Networks

"An event processing network (EPN) is a collection of event processing agents, producers, consumers, and global state elements (...)" (cf. [9], 43). As we have seen in 3.2, most of the elements can be express via components. To combine these components, *Camel* provides two essential elements called *context* and *route*.

Context: The *Camel Context* is a container at runtime level, providing *Camel's* core services, particularly the elements *Registry*, *Type converter*, *Components*, *Endpoints*, *Routes*, *Data formats* and *Languages* (cf. [13], 16).

Route: A *Camel Route* realizes a concrete implementation of a message flow that can be executed on *Camel's* routing engine. It is possible to define multiple routes within a *context*, where each contains a unique identifier (cf. [13], 17).

To get an impression on how a *Camel Context* and *Camel Routes* are structured, we have created a simple listing 4 that provides some clarification. The listing is based upon the Spring DSL (cf. [13], 18) and is embedded in a Spring application context (cf. [18], 27). A *context* element, named *"camelContext"* contains a single *route* with a unique identifier *"camelRoute"*; in addition to that single route it would be possible to add more routes within that context where all can access the same set of core services provided by the *context*. Two core services that must to be registered in our example are the SEDA and JMS component, since they are directly referenced from the route and will be instantiated. Another one is of course the registry that administers all *context paths* for all endpoints to enable a proper endpint resolution when processing the messages.

Listing 4: Context and Route - Application Context

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:camel=
        "http://camel.apache.org/schema/spring"
       xsi:schemaLocation=
        "http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/
             camel-spring.xsd" ... >
  <camel:camelContext id="camelContext">
    <camel:route id="camelRoute">
      <camel:from uri="seda://shipmentOrder"/>
      <camel:to uri=
       "jms:topic:schedule?transacted=true"/>
    </camel:route>
    ...
```

```
14   </camel:camelContext>
15</beans>
```

As we have seen in the example of listing 4, *Camel Context* and *Camel Routes* enable the collection of components (e.g. *event producer*, *consumer* and *channel*). We might want to anticipate the result of section 3.4 to conclude that it is also possible to collect *agents*, *context* and *state* elements within the concept.

The following section explains the concept of *agents*, *context* and *state* in detail and points out concrete implementation strategies for each agent functionality.

3.4 Agents, Context and State

An *event processing agent* is a piece of software that implements the processing logic between an *event producer* and a *consumer* (cf. [9], 42, 51). Each agent provides a specific set of event processing functionality that can be assembled within an *event processing network*. The main tasks of an agent are therefore the mediation between *producer* and *consumer*, particularly the grouping of events according to their processing context. This can be achieved e.g. by *filter*, *split*, *translate*, *aggregate* and *enrich* functionalities.

In addition to these functions, *agents* are also responsible to process *context* and *state* information which enables also the correct routing from the event producing to the event consuming component (cf. [9], 51, 145). The context of an event may have multiple dimensions, e.g. *temporal-*, *spatial-*, or *state-oriented*, where each context dimension can occur in combination with each other one. Identifying the right context for an event can only be conducted, if there is a *global state* element available acting as reference data.

Agent functionality can be reused in different application scenarios. To support the reuse, we have proposed a simplified pattern schema (cf. section 1) containing *name*, *context*, *problem*, *solution* and *example* descriptions.

Even if our intention is not to set a standard, the proposed structure may contribute to the ongoing discussion about standardization within the event processing community (cf. [19]). One of the positive effects a pattern based approach might have is that "[c]learly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces." ([20], 48).

The subsequent paragraphs focus on the agent functionality description for *filtering*, *splitting*, *translation*, *aggregation* and *enrichment* based on the logistics parcel delivery process (cf. figure 1). Even if the industrial background is taken from logistics, we have kept the examples general to enable the adoption towards other industry domains.

3.4.1 Filter Agent Pattern

"A [f]ilter agent (...) performs filtering only and has no matching or derivation steps (...)" ([9], 317). To eliminate uninteresting events it utilizes a *filter expression*. The agent processes events in a stateless manner (cf. [9], 51)

Name: *Filter Agent*

Context: The logistics of parcel delivery is characterized by automated sorting processes. Sorting machines pickup the labeling information required for routing parcels physically to their corresponding gates . While the parcels are being processed on conveyor belts, the capturing of the label information is being conducted based on barcode scans or image recognition. It extracts a huge amount of address data that needs to be distributed to the subsequent delivery nodes.

Problem: Since subsequent processing nodes are only interested in events for their delivery area, it is required to select the particular parcel information and distribute it to the corresponding node. However each node needs the flexibility to change the size of the delivery area, since it must be capable to take over the operation of a nearby delivery area, in case of e.g. low volumes or operational issues in a processing node.

Solution: Providing the flexibility to dynamically change the area, all address events will be published on a single topic. Each node within the delivery network is responsible to create a *filter expression* matching the predefined delivery area. This way it can be assured that each node receives only events for it's field of activity.

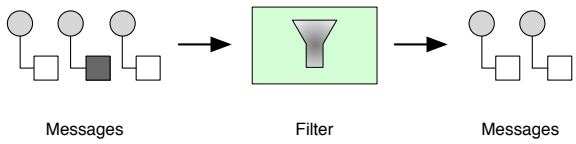

Figure 5: Filter Agent

Example: In the example we are sending address events captured by the sorting machine that contain *name*, *street*, *city* and *zipCode* information. For the purpose of simplicity, we assume that a *zipCode* identifies a section of a delivery area and can not be split into the responsibility of two processing nodes. An active delivery node can serve one to many *zipCode* areas at once while an inactive node maintains zero. As an additional restriction we have defined that a processing node can only serve a coherent, ascending sequence of *zipCodes*. All these restrictions have certainly not a direct, representation in the real parcel delivery process. However they help us to realize a simple *filter expression* as you can see in the in listing 5 by a *predicate*.

The *predicate* evaluates all incoming events for a *minZipCode* and a *maxZipCode* and routes all events within that range to the responsible processing node. *Camel* executes the filtering request within the route by calling the *filter()* expression with a predicate parameter.

Listing 5: Filter Agent

```
1public class FilterRoute extends
2    SpringRouteBuilder {
3  @Override
4  public void configure() throws Exception {
5    Predicate deliveryArea =
6      or(
7        body().isGreaterThan(minZipCode),
8        body().isLessThan(maxZipCode));
9    from("direct://start-body-filter")
10     .filter(deliveryArea)
11     .to("log://after-body-filter?level=INFO")
12     .end();
13  }
14}
```

3.4.2 Split Agent Pattern

A split agent "(...) takes a single incoming event and emits a stream of multiple event objects ()" ([9], 52). It can be used

to partition an incoming event and distribute the individual parts to different consumers (cf. [9], 126).

Name: *Split Agent*

Context: Several customers are having distributed production facilities while maintaining a single administration office that sends out the collected order requests for the entire customer organization. The centralized, batch oriented order processing provides the customer the opportunity to centralize the purchase and accounting department at a single location rather than having multiple employees sitting in each production facility.

Problem: Distributing the bulk of orders is difficult, since the pickup of parcels at the customer's production facilities needs to be executed by different logistics processing nodes. Assigning the individual responsibility for the processing nodes is not possible based on the entire set of order events.

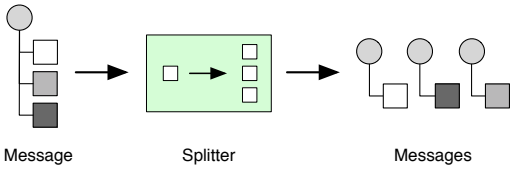

Figure 6: Split Agent

Solution: To enable the responsibility assignment across the logistics network it is required to partition the order collection into single orders where each order contains only the address information of the parcel's sender and recipient. This can be achieved by conducting a split operation on the incoming collection.

Example: The solution can be achieved by splitting the incoming order collection into it's individual parts. *Camel* provides the *split()* expression for conducting those operations. One of the requirements to implement the split expression successfully is to have either a *Collection*, an *Array* or a *NodeList* (cf. [21], 23-24).

Listing 6: Split Agent

```
public class SplitRoute
  extends SpringRouteBuilder {
  @Override
  public void configure() {
    from("direct://start-split")
      .split(body())
      .to("log://split-route?level=INFO");
  }
}
```

3.4.3 Translate Agent Pattern

A translate agent is stateless and "(...) takes a single event as its input, and generates a single derived event which is a function of the input event, using a derivation formula" ([9], 125). Its usage ranges from simple type conversions to complex event transformations modifying event attributes.

Name: *Translate Agent*

Context: The marketing and sales department has figured a way, to extract product data as XML documents from their master spreadsheet where they maintain all product information. They have also found a documentation section from the customer portal provider stateing that there is a Java interface where product data can be updated.

Problem: Unfortunately the Java interface can not serialize XML documents directly and it provides only a remote method invocation interface. The interface is therefore only capable to directly process Java objects. What makes the situation even more complicated is, that some of the extracted attributes do not match to the data structure specified within the interface documentation.

Solution: The only way to support the marketing and sales department in their goal to update the companies portal automatically based on the extracted XML data is to translate the XML product data into the format required by the portal software. It is therefore required to translate, the XML *InputStream* into an object of *Product.class*.

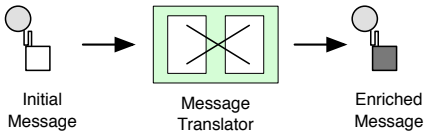

Figure 7: Translate Agent

Example: The XML document that can be extracted from the spreadsheet has a *product* container with the attributes *name*, *size*, *weight* and a *price* (cf. listing 7). Since the specified *Product.class* object has only a description element and no *size* and *weight* attributes, it is required to translate these attributes into a single element.

Listing 7: Extracted XML-based Product

```
<?xml version="1.0" encoding="UTF-8"?>
<Product>
 <Name>Parcel</Name>
 <Size>max. 60x30x15 cm</Size>
 <Weight>up to 2 kg</Weight>
 <Price>4.90</Price>
</Product>
```

The proposed implementation for that translation is a *TypeConverter* (cf. [13], 88-91). Since the default set of *TypeConverter* does not cover the specific translation between the XML-based product document and the Java-based *Product.class*, it is necessary to extend the existing set by a custom converter (cf. listing 8).

Listing 8: Translate Agent - Type Converter

```
@Converter
public class TranslateConverter {
  @Converter
  public Product convertInputStreamToProduct(
      InputStream inputStream) {
    final XPath xPath = XPathFactory.
      newInstance().newXPath();
    final Document document =
      createXMLDocument(inputStream);
    return createProduct(document);
  }
  private Product
    createProduct(Document document) { /*...*/
    return product;
  }
  private Document createXMLDocument(
    InputStream inputStream) { /*...*/
```

```
18     return null;
19   }
20 }
```

To trigger the translation, it is necessary to add a *convertBodyTo(Product.class)* expression to your route. At execution time, the route identifies the object type that reaches at the conversion point and searches for a matching type converter in the *Camel Context* registry.
When the type conversion succeeds, the translate agent sends out the *Product.class* object by executing a remote method invocation (cf. listing 9)

Listing 9: Translate Agent - Route

```
1 public class TranslateRoute
2     extends SpringRouteBuilder {
3   @Override
4   public void configure() {
5     from("direct://start-translate")
6       .convertBodyTo(Product.class)
7       .to("rmi://portal:1099/products");
8   }
9 }
```

3.4.4 Aggregate Agent Pattern

An aggregate agent "(...) takes as input a collection of events and creates a single derived event (...)" ([9], 126). Even if this definition describes the input as *collection* of events it is not within the meaning of a Java*collection* type where a *collection* represents a container object that embraces multiple objects. We want to emphasize that the *aggregate agent* operates on multiple input events and generates a single output event.
When aggregating events, it is sometimes required to execute an aggregation function such as calculating e.g. a sum, an average or the maximum and minimum of a certain event attribute.

Name: *Aggregate Agent*

Context: Since the competition within the parcel delivery market has increased and the prices became more volatile, the marketing and sales department wants to gain an insight into the average price for products and services offered by their competitors.

Problem: Calculating the average market price requires the aggregation of all competitors prices. As soon as one of the competitors changes the price, it is necessary to recalculate the average value.

Solution: To enable the reaction upon any price change, the proposed solution is to aggregate a new average price as soon as one of the competitors updates it's price table. The aggregation will be implemented via an aggregation strategy hiding the complexity of calculating the average value.

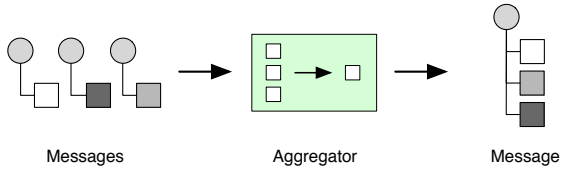

Figure 8: Aggregate Agent

Example: Listing 10 shows the implementation of a *Camel* aggregation strategy. The strategy has as a public *aggregate()* function which takes an old and a new exchange object as parameter values. While the old exchange contains all events that have been collected until the completion criteria is met, the new exchange involves the latest event object. Based on these event objects, it is possible to calculate an average product price and return the calculated value as *exchange* object.

Listing 10: Aggregate Agent - Aggregation Strategy

```
1 public class Aggregator
2     implements AggregationStrategy {
3   private final static List<Product> PRODUCTS =
4     new ArrayList<Product>();
5   public Exchange aggregate(
6       Exchange oldExchange,
7       Exchange newExchange) {
8     final Product oldProduct =
9       getProductFromExchange(oldExchange);
10    final Product newProduct =
11      getProductFromExchange(newExchange);
12    if (null != newAggregateProduct) {
13      if (!PRODUCTS.contains(
14        newAggregateProduct)) {
15          PRODUCTS.add(newAggregateProduct);
16      }
17    }
18    newExchange.getIn().setBody(
19      calcAvgProductPrice());
20    return newExchange;
21  }
22  private Product getProductFromExchange(
23    Exchange exchange) { /* ... */
24    return product;
25  }
26  private double calcAvgProductPrice() {
27    /* ... */
28    return ProductPrice;
29  }
30 }
```

Beside the actual calculation logic and the call of the aggregation strategy (*aggregate()*), the definition of the completion criteria is the most important element that needs to be defined. The criteria defines either the size (*completionSize()*) or the time frame (*completionInterval()* and *completionTimeout()*) that triggers the execution of the aggregation strategy. To assure that the average price is being calculated for every new product event, we have decided set the *completionSize* to one (cf. listing 11).

Listing 11: Aggregate - Agent Route

```
1 public class AggregateRoute
2     extends SpringRouteBuilder {
3   @Override
4   public void configure() {
5     final Aggregator aggregator =
6       new Aggregator();
7     from("direct://start-aggregate")
8       .aggregate(body(), aggregator)
9         .completionSize(1)
10      .to("log://aggregate-route?level=INFO");
11  }
12 }
```

3.4.5 Enrich Agent Pattern

An enrich agent "(...) takes a single input event, uses it to query data from a global state element, and creates a derived

event which includes the attributes from the original event, possibly with modified values, and can include additional attributes." ([9], 126).

Name: *Enrich Agent*

Context: A customer places an order by providing with the physical characteristics and the distance information derived by the points of transfer (e.g. sender and receiver). In contrast to the described order event of section 3.1, he does not offer any product classification that can be consulted for payoff.

Problem: To create an accurate bill, it is required to classify the order into the appropriate product category, e.g *small parcel*, *parcel* or *oversized parcel*, each having an individual price tag.

Solution: To conduct the mapping of the provided information, it is required to lookup the event context in the global state element. This can be achieved by implementing the *enrichment* pattern that implies in the concrete instance the querying of the product database based on the assigned order event to return the corresponding product category.

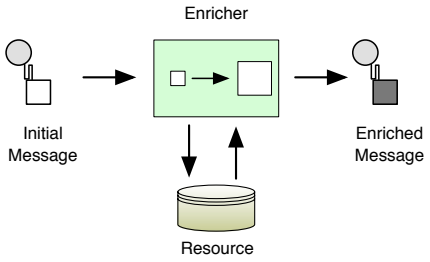

Figure 9: Enrich Agent

Example: The implementation is realized by adopting the aggregation strategy since an aggregation strategy is capable calculating a new output event based on two input exchanges (cf. listing 12).

Listing 12: Enrich Agent - Aggregation Strategy

```
public class Enricher implements
    AggregationStrategy {
  public Exchange aggregate(
      Exchange oldExchange,
      Exchange newExchange) {
    final Context originalContext =
      oldExchange.getIn().getBody(
        Context.class);
    final List<Context> context =
      newExchange.getIn().getBody(List.class);
    /* ... */
    oldExchange.getIn().
      setBody(originalContext);
    return oldExchange;
  }
}
```

What makes the difference compared to a regular aggregation agent is, that one of the exchanges processed within the aggregation strategy is being retrieved by a sub-route that queries the global state. The query is conducted based on a copy of the original incoming event. Listing 13 demonstrates this behavior where the main route calls the sub route when executing the *enrich()* expression.

Listing 13: Enrich Agent - Route

```
public class EnrichRoute
    extends SpringRouteBuilder {
  @Override
  public void configure() {
    configureMainRoute();
    configureEnrichRoute();
  }
  private void configureMainRoute() {
    from("direct://start-enrich")
      .enrich("direct://start-enrichment",
        new Enricher())
      .to("log://enrich-route?level=INFO");
  }
  private void configureEnrichRoute() {
    from("direct://start-enrichment")
    .process(
      new Processor() {
        public void process(
          Exchange exchange) { /* ... */
          exchange.getIn().setBody(context);
        }
      }
    )
    .to("log://enrichment-route?level=INFO");
  }
}
```

4. CONCLUSION

Within this work we have demonstrated, that *Camel* as an integration framework clearly realizes event processing concepts. It provides all the basic requirements needed for *event types and objects* while maintaining an extension mechanism to add additional, upcoming features targeting their structural semantics (cf. section 3.1).

What might be a surprise is, that *Camel's* generic component concept can serve as an abstraction for *event producer, consumer and channel* (cf. section 3.2). Nevertheless, it has been proven that the concept is sound since its adoption can be exemplified with the number of 140 realizing components, all acting either as *event producer*, *consumer* or implementing a *channel*. The introduction to the interaction of event processing networks by realizing *Camel context* and *Camel route* completes the frameworks support for the event processing building blocks.

Finally the paper contributes towards the ongoing standardization effort by introducing the agent functionalities for filtering, splitting, translation, aggregation and enrichment (cf. section 3.4).

Since we have seen, that *Camel* provides such a variety of functions and knowing that it is a lightweight framework, it might be a good candidate to be used for assembling services within a data cloud. An indication supporting this observation is, that *Camel* provides already many components that support the integration of modern technologies and services.

5. REFERENCES

[1] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[2] D. Chandrasekhar. Gartner predictions 2013 for application integration: My take. Technical report, Reality Check, 2013.

[3] W. A. Ruh, F. X. Maginnis, and W. J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley & Sons, 2002.

[4] J. Strachan. Initial checkin of camel routing library, 03 2007.

[5] C. Ibsen. Apache camel was awesome from v1.0 onward. Technical report, DZone, 2013.

[6] The Apache Software Foundation. Apache camel. Technical report, Apache Software Foundation, 2013.

[7] The Apache Software Foundation. Happy 5 years birthday apache camel. Technical report, Apache Software Foundation, 2013.

[8] D. C. Luckham. *Event Processing for Business: Organizing the Real-Time Enterprise*. John Wiley & Sons, 2011.

[9] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2011.

[10] R. v. Ammon, C. Silberbauer, and C. Wolff. Domain specific reference models for event patterns - for faster developing of business activity monitoring applications. In *VIPSI 2007*, October 2007.

[11] C. Emmersberger, F. Springer, and C. Wolff. Location based logistics services and event driven business process management. In D. Tavangarian, T. Kirste, D. Timmermann, U. Lucke, and D. Versick, editors, *Intelligent Interactive Assistance and Mobile Multimedia Computing*, number 53 in Communications in Computer and Information Science, pages 167–177. Springer, 2009.

[12] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, November 2001.

[13] C. Ibsen and J. Anstey. *Camel in Action*. Manning Publications Co., 2011.

[14] N. M. Josuttis. *SOA in Practice - The Art of Distributed System Design*. O'Reilly Media, Inc., first edition edition, 2007.

[15] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. Request for Comments 3986, The Internet Engineering Task Force (IETF), Network Working Group, http://tools.ietf.org/pdf/rfc3986.pdf, January 2005.

[16] M. Welsh. Seda: An architecture for highly concurrent server applications. Project report, Harvard University, http://www.eecs.harvard.edu/ mdw/proj/seda/, May 2006.

[17] M Hapner, R. Burridge, R. Sharma, J. Fialli, and Stout K. Java message service - the jms api is an api for accessing enterprise messaging systems for java programms. Specification Version 1.1, Sun microsystems, April 2002.

[18] Spring. Spring java application framework. Reference Documentation 3.0, Spring Source, http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf, 2011.

[19] R. v. Ammon, C. Emmersberger, T. Ertlmaier, O. Etzion, T. Paulus, and F. Springer. Existing and future standards for event-driven business process management. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, DEBS 2009*. ACM, July 2009.

[20] F. Buschmann. *Pattern oriented software architecture: a system of patters*. John Wiley & Sons, 1st edition, July 1996.

[21] P. Kolb. Realization of eai patterns with apache camel. Studienarbeit 2127, Institut für Architektur von Anwendungssystemen, Universität Stuttgart, 04 2008.